Universidade do Minho
Escola de Engenharia

Sistemas Operativos

Licenciatura em Engenharia Informática

Rastreamento e Monitorização da Execução de Programas

André Lucas Silva Verdelho - a93247

Grupo 97
12 De Maio 2023

1 Introdução

Neste projeto, foi-nos solicitado o desenvolvimento de um sistema de rastreamento de processos. O objetivo era permitir que vários clientes se conectassem ao sistema e enviassem atualizações sobre os processos em execução, sendo que os servidores do sistema deveriam ser capazes de gerir essas atualizações e fornecer informações sobre os processos aos clientes.

Realizei esse projeto com a intenção de cumprir os requisitos básicos e também desenvolver um código relativamente eficiente a nível computacional, e permitindo uma modularização do código no futuro.

Durante o desenvolvimento, utilizei alguns dos conceitos lecionados como named pipes e forks para a comunicação entre cliente e servidor, bem como estruturas de dados simples e funções do sistema para manipulação de processos e monitorização de tempo.

2 Tracer

A função principal **main** deste programa recebe os argumentos da linha de comandos e realiza diferentes ações com base nesses argumentos. A função inicia verificando o número de argumentos passados para determinar a opção selecionada e, se, e o número de argumentos for inferior a 2, uma mensagem a demonstrar o uso correto é exibida. Em seguida, o programa é encerrado com o código de saída 1. Caso contrário, uma opção foi passada como input pelo que é agora necessário confirmar se é válida. O programa dispõe de três comandos principais:

- **-u <programa> [<args>...]** - Executa o programa num processo filho e notifica o servidor aquando do seu principio e fim.
- **-s** - Requisita ao servidor toda a informação relativa a execuções de programas (terminadas e não) e imprime para o **STDOUT** a resposta recebida.
- **-q** - Termina a execução do programa

Em seguida, a opção selecionada é obtida do segundo argumento passado **argv[1]**. Se a opção for **"-u"**, o **tracer** irá executar o programa especificado. Neste caso, é verificado se existe pelo menos mais um argumento após a opção, que corresponderia ao nome do programa a ser executado. Se não houver, uma mensagem de erro é exibida indicando que o nome do programa não foi dado.

O nome do programa é armazenado na variável **program_name**, e o número de argumentos do programa é calculado subtraindo 3 do número total de argumentos passados como input (argc). Um array de pointers para strings **program_args** é criado com tamanho igual ao número de argumentos mais 2.

O primeiro elemento do array **program_args[0]** guarda o nome do programa. Em seguida, os restantes argumentos são guardados no array recorrendo a um ciclo que itera sobre os argumentos de input. No final do loop, o último elemento do array **program_args[num_args + 1]** é definido como NULL para indicar o final da lista de argumentos.

Em seguida, a função **fork()** é chamada para criar um novo processo. Se fork() retornar -1, ocorreu um erro na criação do processo e uma mensagem de erro é exibida usando a função **perror()**. Se retornar 0, significa que o processo é o filho. Nesse caso, a função **execute_program()** é chamada, passando o nome do programa e os argumentos como parâmetros. Em seguida, o processo filho é encerrado com o código de saída 0.

Se a chamada a fork() retornar um valor diferente de 0 e -1, significa que o processo é o pai. Nesse caso, a execução continua para a próxima iteração do loop.

Se a opção selecionada for "-s", a função **query_running_programs()** é chamada para consultar o status dos programas ao servidor.

Se a opção selecionada for "-q", o loop principal é encerrado e o programa é finalizado.

Se for passada qualquer outra opção, esta é ignorada e é iniciado o ciclo principal da função que funciona como a interface entre o utilizador e o programa Tracer. Neste ciclo são utilizadas as mesmas flags que as do inicio do programa e a lógica da execução dos programas é idêntica.

Figure 1: Exemplo de funcionamento do **tracer**

2.1 execute_program()

A função **execute_program** é responsável por executar um programa e enviar notificações ao programa **monitor** relacionadas à execução desse programa.

Primeiro, é verificado se o **FIFO** (named pipe) em **TRACER_FIFO_PATH** existe. Se não existir, é criado utilizando a função **mkfifo** e, em caso de erro na sua criação, uma mensagem de erro é exibida e o processo é encerrado.

De seguida, o **FIFO** é aberto para escrita utilizando a função open, e em caso de falha na abertura, uma mensagem de erro é exibida e o processo é encerrado.

A função **getpid** é chamada para obter o ID do processo atual e essa informação é usada para construir uma mensagem de notificação, informando o utilizador sobre o nome do programa que vai ser executado e o seu **PID**. Essa mensagem é escrita no terminal usando write para o **STDOUT**.

O tempo de início da execução do programa é obtido utilizando a função **gettimeofday** e armazenado na estrutura **start_time**. Em seguida, uma mensagem contendo o **PID**, o nome do programa e o tempo de início é formatada e escrita no **FIFO**. Essa mensagem é destinada ao **monitor** (servidor) e serve como notificação de que um programa está prestes a ser executado.

Num novo processo, criado utilizando fork, o programa é executado utilizando a função **execvp**. Caso ocorra algum erro na execução, uma mensagem de erro é exibida utilizando a função perror. Em seguida, o tempo de término da execução é obtido usando **gettimeofday** e armazenado no **finish_time**.

Se a execução do programa resultar em erro uma nova mensagem é formatada, contendo o **PID**, a indicação de que o programa terminou (**finished**) e o **finish_time**, e é escrita no **FIFO** e é terminado o processo.

Se o processo for bem-sucedido, o processo pai espera pelo término do processo filho utilizando a função wait e envia uma mensagem para o servidor confirmando o fim do processo e um timestamp.

2.2 query_running_programs()

A função **query_running_programs** é responsável por enviar uma mensagem ao servidor requisitando o status dos programas que já foram executados e em execução, bem como a sua duração e tempo total de execução.

Primeiro, o **FIFO** no **TRACER_FIFO_PATH** é aberto para escrita utilizando a função open. Em caso de falha na abertura, uma mensagem de erro é exibida e o programa é encerrado.

O ID do processo atual é obtido utilizando getpid e armazenado na variável pid. Uma mensagem contendo o **PID** e a indicação de status é formatada e escrita no **FIFO** utilizando a função write.

O **tracer_fifo** é fechado utilizando close e de seguida, é verificado se o **monitor_fifo** em **MONITOR_FIFO_PATH** existe. Se não existir, ele é criado utilizando a função mkfifo e em caso de erro ao criar o FIFO, uma mensagem de erro é exibida e o programa é encerrado.

O **monitor_fifo** é aberto para leitura utilizando a função open e em caso de falha na abertura, uma mensagem de erro é exibida e o programa é encerrado.

As entradas são lidas utilizando a função read e impressas no **STDOUT** utilizando a função write. O loop continua até que não existam mais bytes para ler. Por fim, o **monitor_fifo** é fechado utilizando close.

3 Monitor

A função **main** no programa monitor.c é responsável por ler as atualizações enviadas pelo programa cliente através do **FIFO** e executar as operações correspondentes.

Primeiro, é verificado se o **FIFO** em **TRACER_FIFO_PATH** existe. Se não existir, é criado utilizando a função mkfifo e em caso de erro ao criar o FIFO, uma mensagem de erro é exibida e o programa é encerrado.

O FIFO é aberto para leitura e escrita utilizando a função open. Em caso de falha na abertura, uma mensagem de erro é exibida e o programa é encerrado. De seguida, inicia-se um loop infinito que espera por atualizações do cliente. A query do cliente é lida do FIFO utilizando a função read e armazenada na variável **buffer**. Em caso de erro ao ler do FIFO, uma mensagem de erro é exibida e o programa é encerrado.

A requisição do cliente é então analisada e os campos **PID, informação/status e timestamp** são extraídos do buffer utilizando a função sscanf. O timestamp é convertido para a estrutura timeval. A query recebida é exibida no terminal para ajudar no processo de debug, utilizando a função printf. Dependendo do valor da informação/status recebida, diferentes ações são executadas:

- Se a informação/status for **"finished"**, a função **update_entry** é chamada para atualizar a entrada correspondente com um sinal de término de programa.
- Se a informação/status for **"status"**, a função **print_entries** é chamada para imprimir as entradas armazenadas.
- Caso contrário, a função **insert_entry** é chamada para inserir uma nova entrada na estrutura de dados.

O buffer leva com um *"reset"* para evitar resíduos de dados anteriores. O loop continua a esperar por atualizações do cliente. Por fim, o FIFO é fechado utilizando a função close.

```c
#define MAX_ENTRIES 256

typedef struct
{
    int pid;
    char command[256];
    struct timeval start_time;
    struct timeval end_time;
} ExecutionInfo;

ExecutionInfo executionInfos[MAX_ENTRIES];
int numEntries = 0;
```

Figure 2: Estruturas de dados usadas para armazenar as informações dos programas

3.1 insert_entry() e update_entry()

A função **insert_entry** é responsável por inserir uma nova entrada na estrutura de dados **executionInfos**. Recebe como argumentos o **PID** do processo, o comando executado e o tempo de início.

Primeiro, é verificado se o número máximo de entradas foi atingido. Se isto acontecer, uma mensagem é exibida informando que o número máximo de entradas foi alcançado e a função retorna.

Em seguida, a entrada é adicionada na próxima posição disponível no array executionInfos. O PID é atribuído ao campo pid da entrada. O comando é copiado para o campo command da entrada, utilizando a função strncpy e o tempo de início é atribuído ao campo **start_time** da entrada. Por fim, a variável global de número de entradas é incrementada.

A função **update_entry** é responsável por atualizar uma entrada existente na estrutura de dados execution-Infos. Recebe como argumentos o PID do processo, o status de saída e o tempo de término.

A função percorre todas as entradas existentes no array executionInfos para encontrar a entrada correspondente ao **PID** fornecido. Se a entrada for encontrada, é atualizado o valor do campo **end_time**, notificado o servidor para debug e terminada a execução desta função através do return.

Se a entrada não for encontrada no array executionInfos, é exibida uma mensagem informando que a entrada correspondente ao PID fornecido não foi encontrada.

3.2 print_entries()

Esta função é responsável por enviar para o cliente toda a informação necessária, neste caso, todos os programas que já foram executados através do cliente, assim como a sua duração. É enviado um por linha seguindo o padrão definido como : **PID;PROGRAM;DURATION** e no final de todas as entradas é também enviada uma linha adicional com o tempo acumulado de todos os processos.

Para tal é verificado de o **monitor_fifo** existe e é criado caso contrário. De seguida é aberto para escrita e através de um **for loop** é lida cada entrada do array que contem toda a informação dos processos e calculado o tempo de execução se já tiver terminado o programa e o tempo em execução caso contrário. Após terem sido calculadas as durações, são enviadas as entradas seguindo o padrão para o FIFO que vai ser aberto para leitura pelo cliente, somando também a um acumulador **total_duration** a duração do programa.

4 Falhas e Melhorias

Após terminar a minha solução para o trabalho fiquei insatisfeito com o código que desenvolvi e decidi analisar os vários aspetos que poderiam ter sido facilmente corrigidos e até melhorados caso tivesse começado a realizar o trabalho com mais antecedência.

4.1 Monitor e as suas funções

O servidor que desenvolvi é pouco complexo num sentido negativo - não cria processos para lidar com clientes diferentes paralelamente, recorrendo apenas às duas fifos estabelecidas para comunicar com um cliente singular ou com vários mas sem ser feita qualquer distinção de quais programas foram executados por qual cliente.

Para resolver este problema poderia ter implementado algumas soluções, sendo que a que tentei implementar mas acabei por não conseguir pôr em funcionamento a tempo envolvia a criação de um processo filho após o servidor realizar com o cliente uma interação semelhante a um "handshake", onde eram definidos os fifos pelos quais iriam comunicar. Atribuindo um inteiro aleatório ao cliente e concatenando esse client_id ao path que defini para os fifos principais, eram deste modo definidos fifos privados para cada cliente comunicar com um processo filho do servidor.

Outra alteração que também teria de ser feita era à estrutura de dados na qual era armazenada a informação. Poderia utilizar uma HashTable de listas ligadas onde a key seria o id do cliente e o valor a lista com todos os programas respetivos. Dada a escala reduzida deste projeto seria a solução ideal para o problema definido.

5 Conclusões

Tendo finalizado o trabalho, após não ter realizado um trabalho de acordo com os meus padrões de qualidade, estou satisfeito com a jornada e o progresso que fiz durante toda a realização do projeto. Acredito que foi um trabalho essencial no qual desenvolvi o meu espírito crítico e autonomia.

Contudo é importante notar o contexto em que desenvolvi o trabalho. O contacto que tive com C até este projeto foi limitado pela simplicidade dos trabalhos que já tinha desenvolvido, e sinto que aprofundei bastantes conceitos essenciais ao desenvolver a solo o projeto. Como uma boa marca de um excelente enunciado, acredito que os conhecimentos com que fiquei após o seu desenvolvimento me vão levar a produzir melhores trabalhos no futuro.